BIAX Corporation v. Intel
Civil Action No. 2:05-cv-184-TJW

EXHIBIT 2
(PART 2)

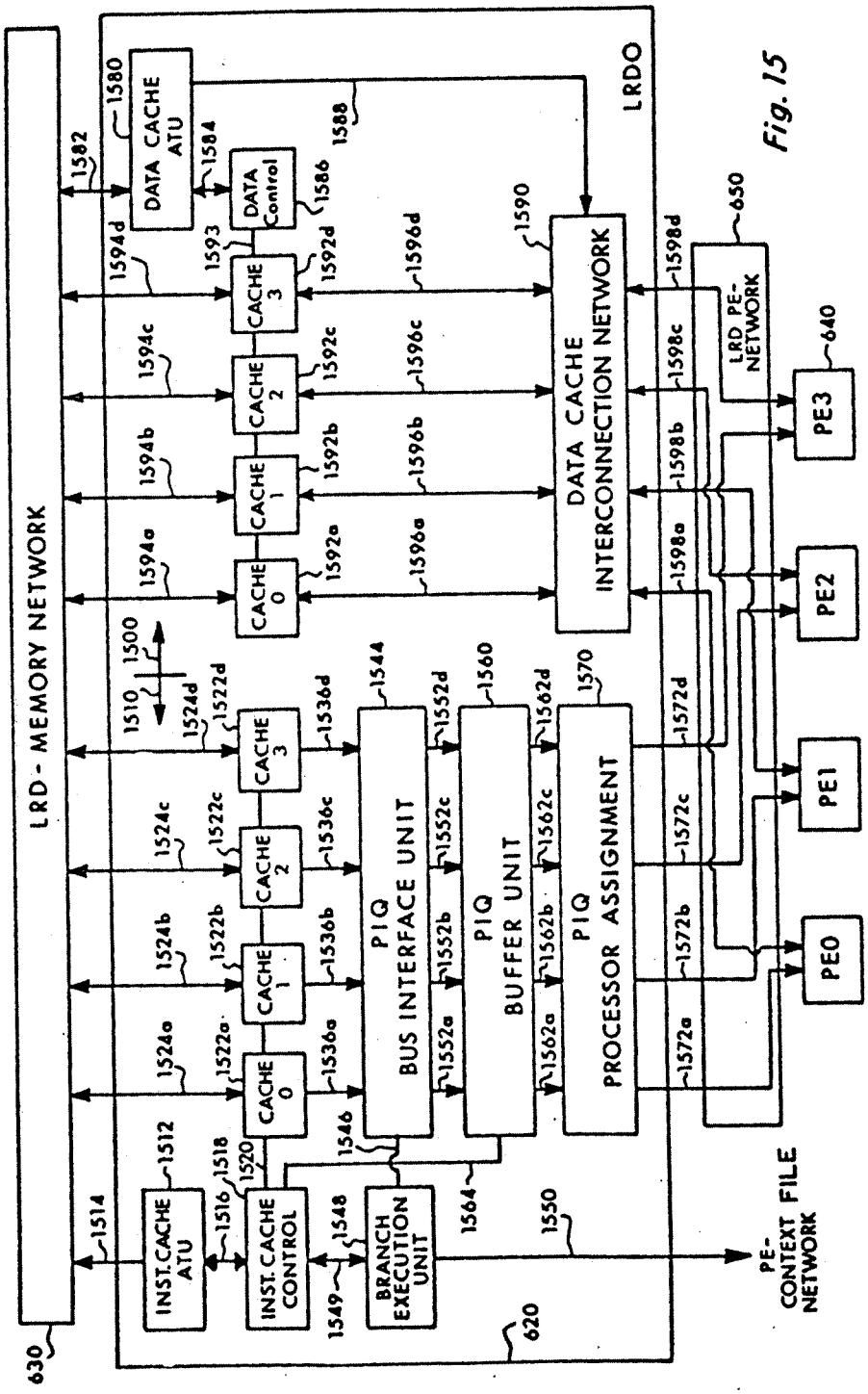


Fig. 15

BIA0001135

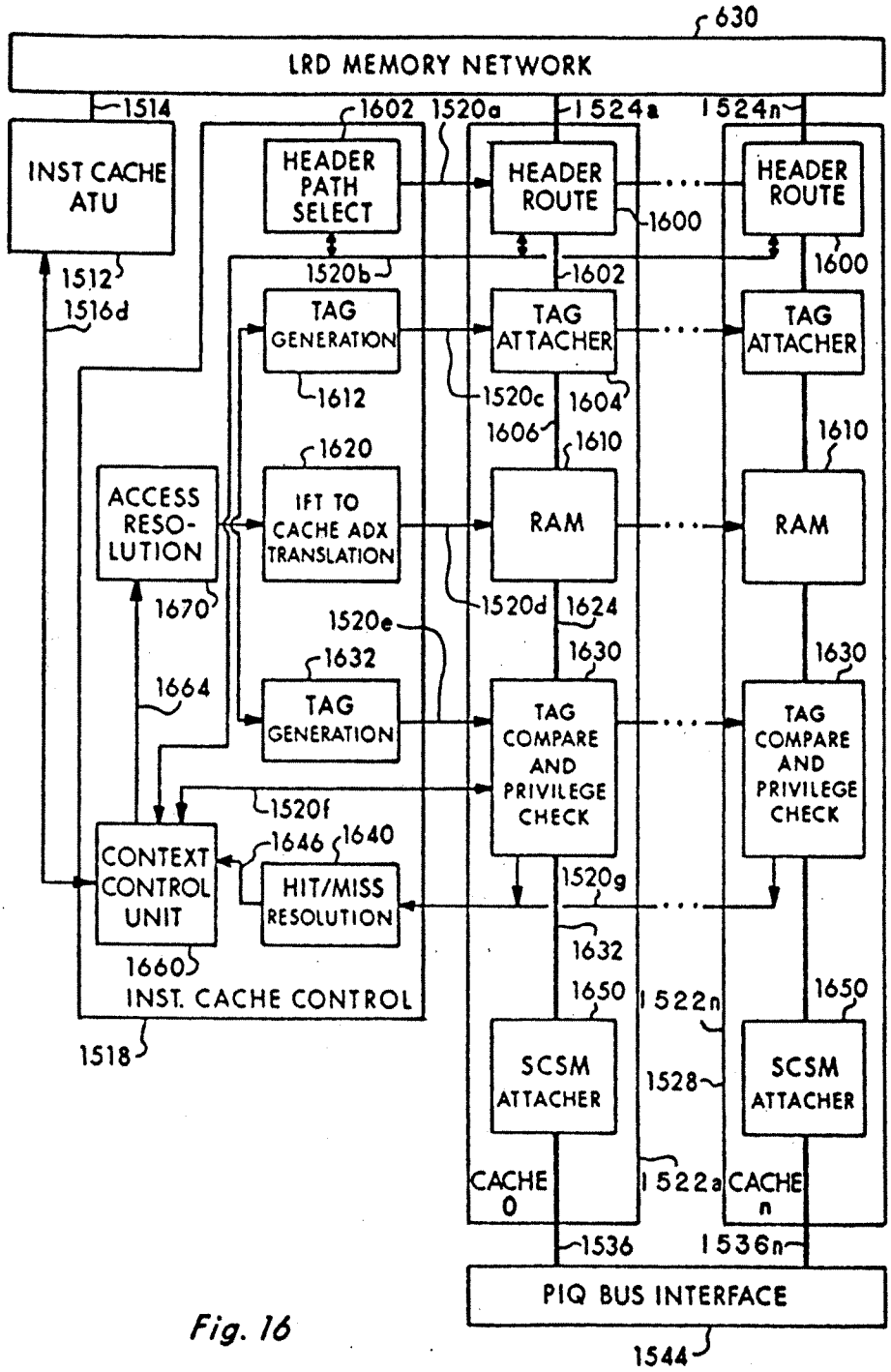


Fig. 16

BIA0001136

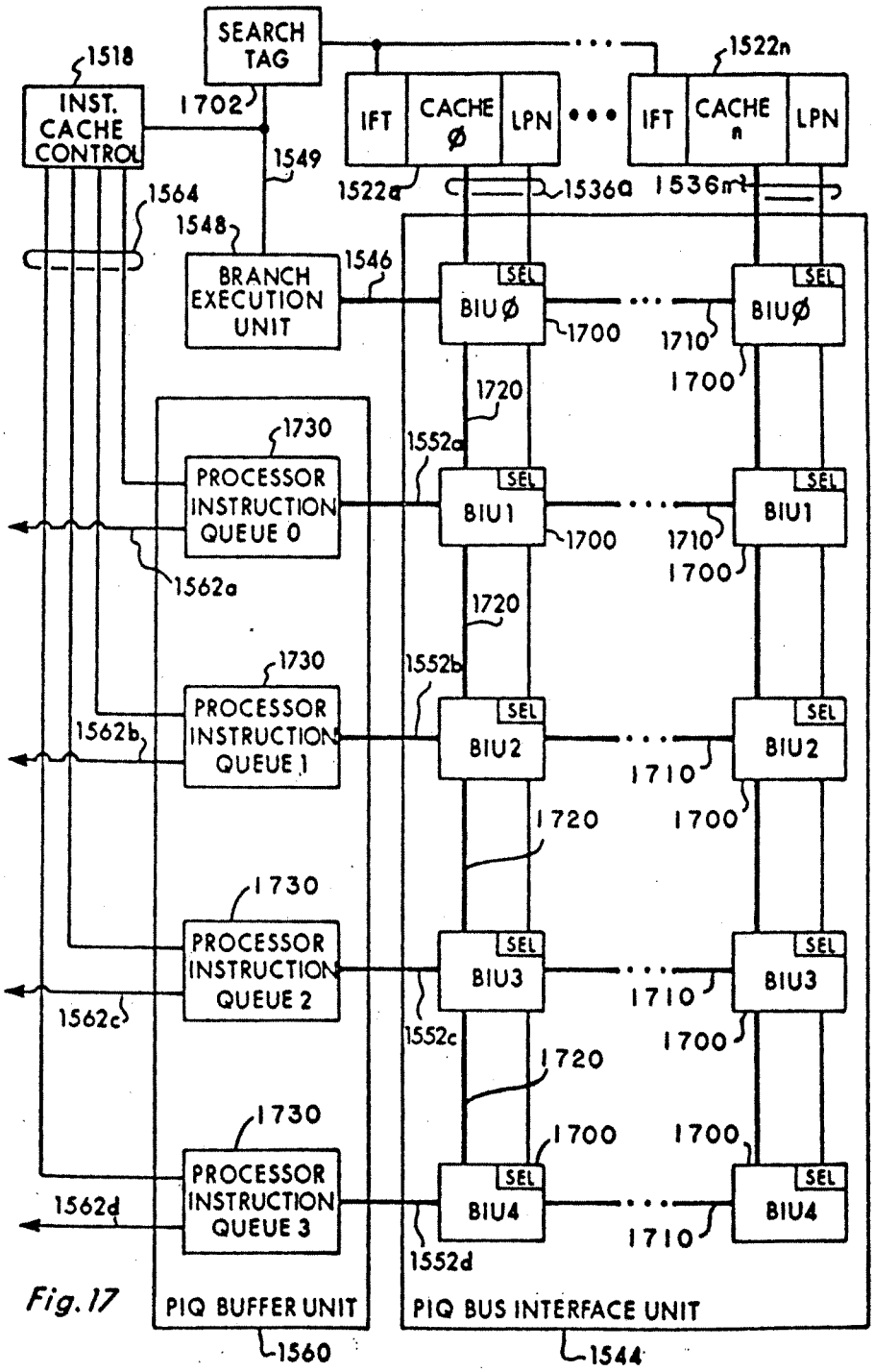
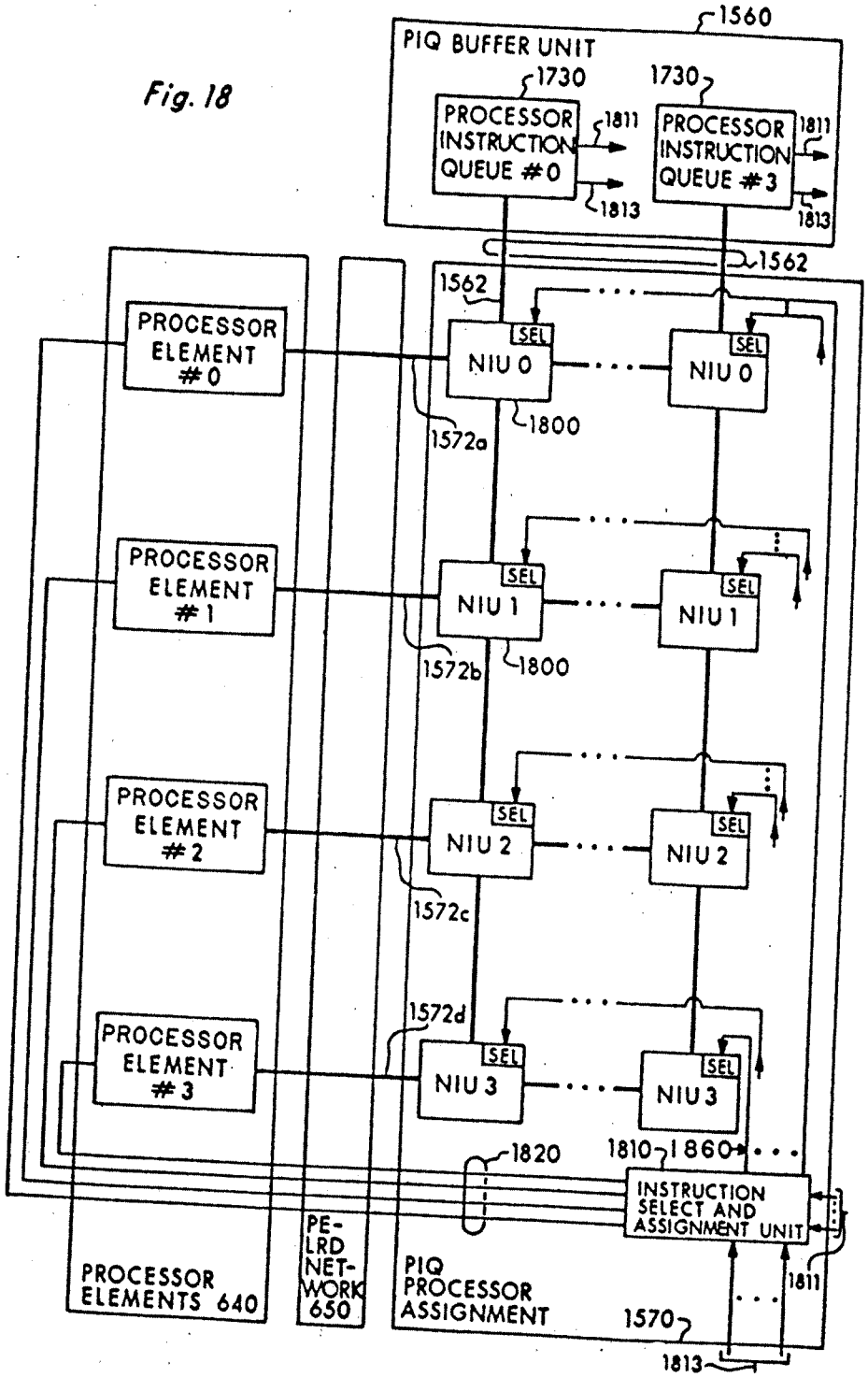


Fig. 17

BIA0001137

Fig. 18



BIA0001138

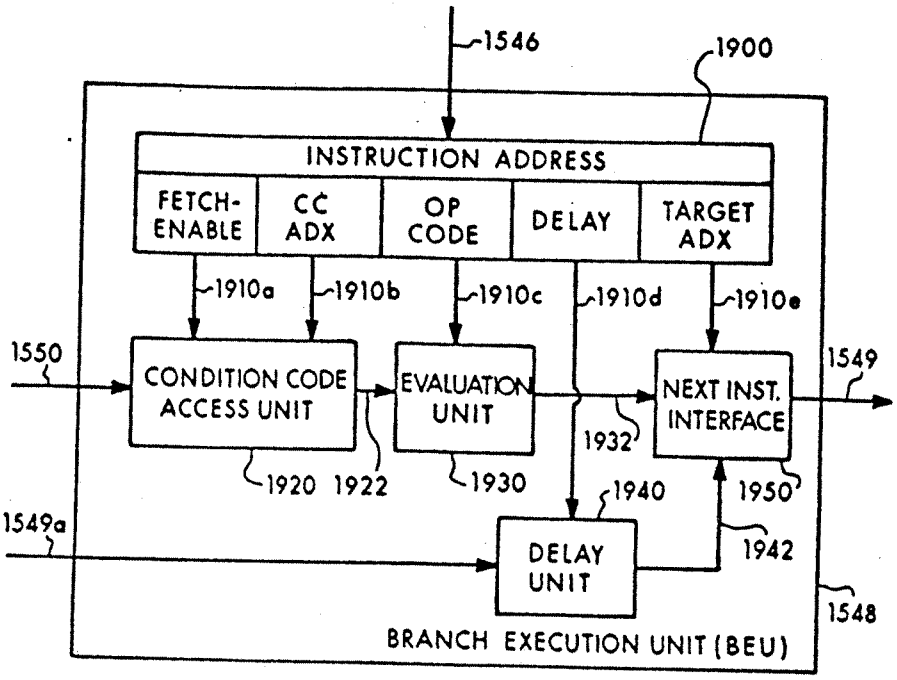


Fig. 19

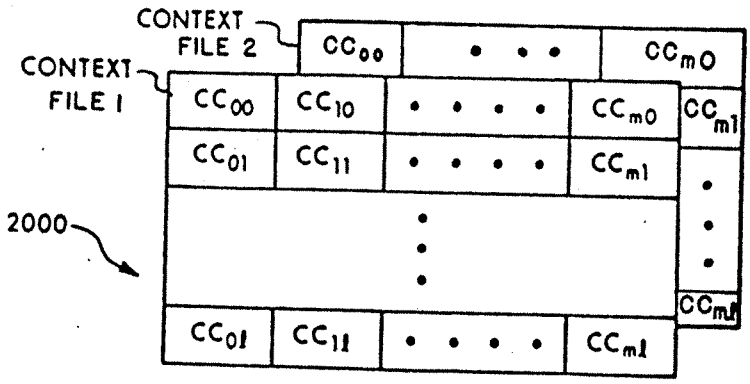


Fig. 20

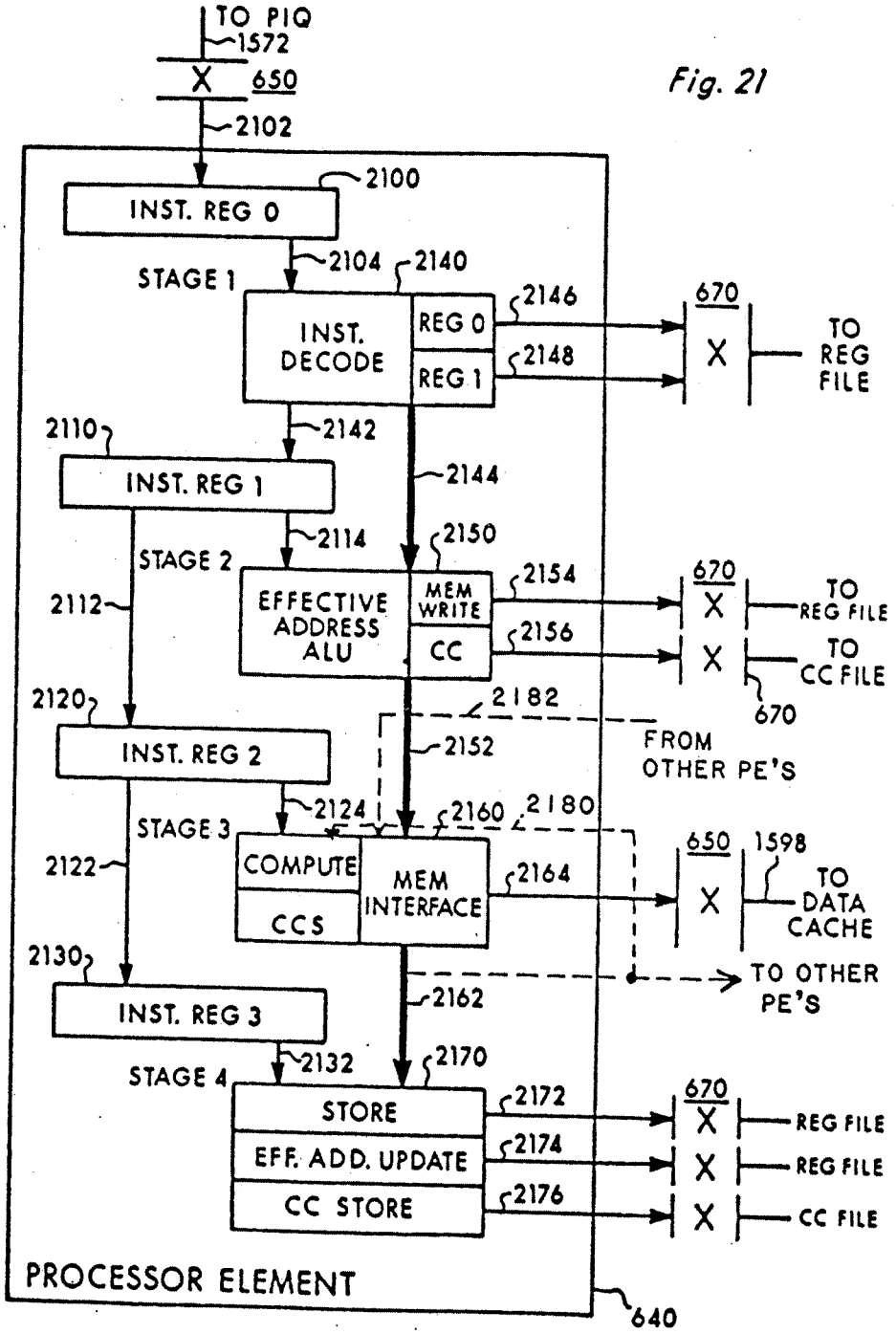


Fig. 21

BIA0001140

Fig. 22a

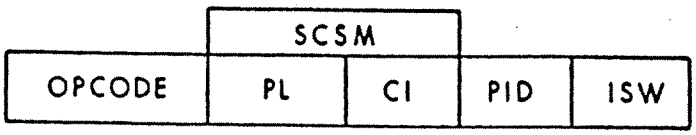


Fig. 22b

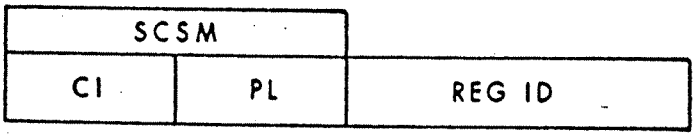


Fig. 22c

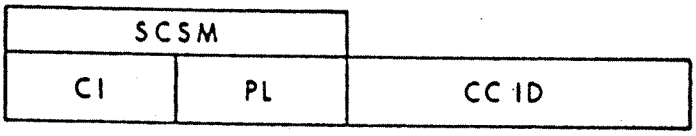
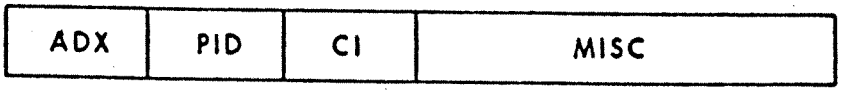


Fig. 22d



BIA0001141

PARALLEL PROCESSOR SYSTEM FOR PROCESSING NATURAL CONCURRENCIES AND METHOD THEREFOR

This is a division of application Ser. No. 794,221, filed Oct. 31, 1985, now U.S. Pat. No. 4,847,755, issued July 11, 1989.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention generally relates to parallel processor computer systems and, more particularly, to parallel processor computer systems having software for detecting natural concurrencies in instruction streams and having a plurality of identical processor elements for processing the detected natural concurrencies.

2. Description of the Prior Art

Almost all prior art computer systems are of "Von Neumann" construction. In fact, the first four generations of computers are Von Neumann machines which use a single large processor to sequentially process data. In recent years, considerable effort has been directed towards the creation of a fifth generation computer which is not of the Von Neumann type. One characteristic of the so-called fifth generation computer relates to its ability to perform parallel computation through use of a number of processor elements. With the advent of very large scale integration (VLSI) technology, the economic cost of using a number of individual processor elements becomes cost effective.

An excellent survey of multi-processing technology involving the use of parallel architectures is set forth in the June, 1985 issue of COMPUTER, published by the IEEE Computer Society, 345 East 47th Street, New York, New York 10017, which is not believed to be prior art to the present invention but serves to define the activity in this field. In particular, this issue contains the following articles:

(a) "Essential Issues in Multi-processor Systems" by Gajski et al.;

(b) "Microprocessors: Architecture and Applications" by Patton; and

(c) "Research on Parallel Machine Architecture for Fifth-Generation Computer Systems" by Murakami et al.

In addition, an article in the IEEE Spectrum, November, 1983, entitled "Computer Architecture" by A. L. Davis discusses the features of such fifth-generation machines. Whether or not an actual fifth generation machine has yet been constructed is subject to debate, but various features have been defined and classified.

As Davis recognizes, fifth-generation machines should be capable of using multiple-instruction, multiple-data (MIMD) streams rather than simply being a single instruction, multiple-data (SIMD) system. The present invention is believed to be of the fifth-generation non-Von Neumann type which is capable of using MIMD streams in single context (SC-MIMD) or in multiple context (MC-MIMD). The present invention, however, also finds application in the entire computer classification of single and multiple context SIMD (SC-SIMD and MC-SIMD) machines as well as single and multiple context single-instruction, single data (SC- o SISD and MC-SISD) machines.

While the design of fifth-generation computer systems is fully in a state of flux, certain categories of systems have been defined. Davis and Gajski, for example, base the type of computer upon the manner in which

"control" or "synchronization" of the system is performed. The control classification includes control-driven, data-driven, and reduction (or demand) driven. The control-driven system utilizes a centralized control such as a program counter or a master processor to control processing by the slave processors. An example of a control-driven machine is the Non-Von-1 machine at Columbia University. In data-driven systems, control of the system results from the actual arrival of data required for processing. An example of a data-driven machine is the University of Manchester dataflow machine developed in England by Ian Watson. Reduction driven systems control processing when the processed activity demands results to occur. An example of a reduction processor is the MAGO reduction machine being developed at the University of North Carolina, Chapel Hill. The characteristics of the non-Von-1 machine, the Manchester machine, and the MAGO reduction machine are carefully discussed in the Davis article. In comparison, data-driven and demand-driven systems are decentralized approaches whereas control-driven systems are centralized. The present invention is more properly categorized in a fourth classification which could be termed "time-driven." Like data-driven and demand-driven systems, the control system of the present invention is decentralized. However, like the control-driven system, the present invention conducts processing when an activity is ready for execution.

It has been recognized by the Patton, Id. at 39, that most computer systems involving parallel processing concepts have proliferated from a large number of different types of computer architectures. In such cases, the unique nature of the computer architecture mandates or requires either its own processing language or substantial modification of an existing language to be adapted for use. To take advantage of the highly parallel structure of such computer architectures, the programmer is required to have an intimate knowledge of the computer architecture in order to write the necessary software. As a result, porting programs to these machines requires substantial amounts of the users effort, money and time.

Concurrent to this activity, work has also been progressing on the creation of new software and languages, independent of a specific computer architecture, that will expose (in a more direct manner), the inherent parallelism. Hence, most effort in designing supercomputers has been concentrated at the hardware end of the spectrum with some effort at the software end.

Davis has speculated that the best approach to the design of a fifth-generation machine is to concentrate efforts on the mapping of the concurrent program tasks in the software onto the physical hardware resources of the computer architecture. Davis terms this approach one of "task-allocation" and tauts it as being the ultimate key to successful fifth-generation architectures. He categorizes the allocation strategies into two generic types. "Static allocations" are performed once, prior to execution, whereas "dynamic allocations" are performed by hardware whenever the program is executed or run. The present invention utilizes a static allocation strategy and provides task allocations for a given program after compilation and prior to execution. The recognition of the "task allocation" approach in the design of fifth generation machines was used by Davis in the design of his "Data-driven Machine-II" constructed at the University of Utah. In the Data-driven Machine-II, the program was compiled into a program

graph that resembles the actual machine graph or architecture.

Task allocation is also referred to as "scheduling" in the Gajski article. Gajski sets forth levels of scheduling to include high level, intermediate level, and low level scheduling. The present invention is one of low-level scheduling, but it does not use conventional scheduling policies of "first-in-first-out", "round-robin", "shortest type in job-first", or "shortest-remaining-time." Gajski also recognizes the advantage of static scheduling in that overhead costs are paid at compile time. However, Gajski's recognized disadvantage, with respect to static scheduling, of possible inefficiencies in guessing the run time profile of each task is not found in the present invention. Therefore, the conventional approaches to low-level static scheduling found in the Occam language and the Bulldog compiler are not found in the software portion of the present invention. Indeed, the low-level static scheduling of the present invention provides the same type, if not better, utilization of the processors commonly seen in dynamic scheduling by the machine at run time. Furthermore, the low-level static scheduling of the present invention is performed automatically without intervention of programmers as required (for example) in the Occam language.

Davis further recognizes that communication is a critical feature in concurrent processing in that the actual physical topology of the system significantly influences the overall performance of the system.

For example, the fundamental problem found in most data-flow machines is the large amount of communication overhead in moving data between the processors. When data is moved over a bus, significant overhead, and possible degradation of the system, can result if data must contend for access to the bus. For example, the Arvind data-flow machine, referenced in Davis, utilizes an I-structure stream in order to allow the data to remain in one place which then becomes accessible by all processors. The present invention teaches a method of hardware and software based upon totally coupling the hardware resources thereby significantly simplifying the communication problems inherent in systems that perform multiprocessing.

Another feature of non-Von Neumann type multiprocessor systems is the level of granularity of the parallelism being processed. Gajski terms this "partitioning." The goal in designing a system according to Gajski is to obtain as much parallelism as possible with the lowest amount of overhead. The present invention performs concurrent processing at the lowest level available, the "per instruction" level. The present invention teaches a method whereby this level of parallelism is obtainable without execution time overhead.

Despite all of the work that has been done with multiprocessor parallel machines, Davis (Id. at 99) recognizes that such software and/or hardware approaches are primarily designed for individual tasks and are not universally suitable for all types of tasks or programs as has been the hallmark with Von Neumann architectures. The present invention sets forth a computer system that is generally suitable for many different types of tasks since it operates on the natural concurrencies existent in the instruction stream at a very fine level of granularity.

The patent invention, therefore, pertains to a non-Von Neuman MIMD computer system capable of operating upon many different and conventional programs by a number of different users. The natural concurrencies in each program are statically allocated, at a very fine level of granularity, and intelligence is added to each instruction at essentially the object code level. The added intelligence includes a logical processor number and an instruction firing time in order to provide the time-driven decentralized control for the present invention. The detection and low level scheduling of the natural concurrencies and the adding of the intelligence occurs only once for a given program, after conventional compiling of the program, without user intervention and prior to execution. The results of this static allocation are executed on a system containing a plurality of identical processor elements. These processor elements are characterized by the fact that they contain no execution state information from the execution of previous instructions, i.e., they are context free. In addition, a plurality of contexts, one for each user, are provided wherein the plurality of context free processor elements can access any storage resource contained in any context through total coupling of the processor element to the shared resource during the processing of an instruction. Under the teachings of the present invention no condition code or results registers are found on the individual processor elements.

Based upon the features of the present invention, a patentability investigation was conducted resulting in the following references:

ARTICLES

Dennis, "Data Flow Supercomputers", Computer, November, 1980, Pgs. 48–56.

Hagiwara, H. et al., "A Dynamically Microprogrammable, Local Host Computer With Low-Level Parallelism", IEEE Transactions on Computers, C-29, No. 7, July, 1980, Pgs. 577–594.

Fisher et al., "Microcode Compaction: Looking Backward and Looking Forward", National Computer Conference, 1981, Pgs. 95–102.

Fisher et al., "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs", IEEE No. 0194-1895/81/0000/0171, 14th Annual Microprogramming Workshop, Sigmicro, October, 1981, Pgs. 171–182.

J.R. Vanaken et al., "The Expression Processor, IEEE Transactions on Computers, C-30, No. 8, August, 1981, Pgs. 525–536.

Bernhard, "Computing at the Speed Limit", IEEE Spectrum, July, 1982, Pgs. 26–31.

Davis, "Computer Architecture", IEEE Spectrum, November, 1983, Pgs. 94–99.

Hagiwara, H. et al., "A User-Microprogrammable Local Host Computer With Low-Level Parallelism", Article, Association for Computing Machinery, #0149-7111/83/0000/0151, 1983, Pgs. 151–157.

McDowell, Charles Edward, "SIMAC: A Multiple ALU Computer", Dissertation Thesis, University of California, San Diego, 1983 (111 pages).

McDowell, Charles E., "A Simple Architecture for Low Level Parallelism", Proceedings of 1983 International Conference on Parallel Processing, Pgs. 472–477.

Requa, et al., "The Piecewise Data Flow Architecture: Architectual Concepts, IEEE Transactions on Computers, Vol. C-32, No. 5, May, 1983, Pgs. 425–438.

Fisher, A.T., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", Computer, 1984, Pgs. 45–52.

Fisher, et al., "Measuring the Parallelism Available for Very Long Instruction, Word Architectures",

IEEE Transactions on Computers, Vol. C-33, No. 11, November, 1984, Pgs. 968–976.

PATENTS

Freiman, et al., (3,343,135), "Compiling Circuitry for a Highly-Parallel Computing System", Sept. 19, 1967.

Reigel, (3,611,306), "Mechanism to Control the Sequencing of Partially Ordered Instructions in a Parallel Data Processing System", Oct. 5, 1971.

Culler, (3,771,141), "Data Processor With Parallel Operations for Instructions", Nov. 6, 1973.

Gruner, (4,104,720), "CPU/Parallel Processor Interface with Microcode Extension", Aug. 1, 1978.

Blum, et al. (4,109,311), "Instruction Execution Modification Mechanism for Time Slice Controlled Data Processors", Aug. 22, 1978.

Dennis, et al. (4,153,932), "Data Processing Apparatus for Highly Parallel Execution of Stored Programs", May 8, 1979.

Kober, (4,181,936), "Data Exchange Processor for Distributed Computing System", Jan. 1, 1980.

Bernhard, (4,228,495), "Multiprocessor Numerical Control System", Oct. 14, 1980.

Gilliland, et al., (4,229,790), "Concurrent Task and Instruction Processor and Method", Oct. 21, 1980.

Caril, (4,241,398), "Computer Network Line Protocol System", Dec. 23, 1980.

Koehler, et al., (4,270,167), "Apparatus and Method for Cooperative and Concurrent Coprocessing of Digital Information", May 26, 1981.

Lorie, et al., (4,435,758), "Method for Conditional Branch Execution in SIMD Vector Processors", Mar. 6, 1984.

DeSantis, (4,468,736), "Mechanism for Creating Dependency Free Code For Multiple Processing Elements", Aug. 28, 1984.

DeSantis, (4,466,061), "Concurrent Processing Elements For Using Dependency Free Code", Aug. 14, 1984.

In the two McDowell references, a parallel processing system utilizing low level parallelism was disclosed. McDowell differentiates between low level and high level parallelism in that low level parallelism pertains to the execution of two or more machine level operations in parallel whereas high level parallelism is the parallel execution of high level language constructs which includes source language statements, tasks, procedures, or entire programs.

Like the present invention, McDowell identifies and statically schedules low level parallel operations found in existing "sequential" programming languages. Further, he does not require the creation of special purpose programming languages or user modifications to the parallelism existing with the program at compile time, rather he simply analyzes the low level parallelism existing within the basic blocks (BBs) that make up the program.

In the McDowell SIMAC system, processor elements are identical and communicate through a shared memory and a set of shared registers. The processor elements are controlled by a control processor that is fed by a single instruction stream.

McDowell recognizes his particular parallel processing system does not fit into the standard categories of: homogeneous multiprocessors, non-homogeneous multi-processors, array processors, and pipeline processors. Rather, McDowell classified his computer system in the class of Schedulable Parallel Instruction Execution (SPIE) computers. McDowell's reasoning as to why SIMAC is categorized as a SPIE processor is as follows:

> SPIE processors have only one instruction stream and are therefore immediately eliminated from the first two groups. They have some of the characteristics of array processors, but they are distinctly different in that each of the several processing elements may be executing different operations. In array processors all processing elements are restricted to executing the same instruction on different data. SPIE processors are also similar in capability to pipelined machines but again they are distinct in that each processing element may be given a new operation each instruction cycle that is independent of the operation it was performing in the previous cycle. Pipelined machines only specify a single operation per instruction cycle, and then other operations on other processing elements may be triggered from that one initial operation. Dissertation Thesis at 11.

The present invention also is properly categorized as a SPIE processing system since each processor element may be given a new operation each instruction cycle that is independent of the operation it was performing in the previous cycle. Hence, the McDowell reference provides important background material to the teachings of the present invention. However, important distinctions exist between McDowell's SIMAC system and the system of the present invention. SIMAC is quoted as being a SPIE machine that can perform scalar concurrent execution of a single user's program (i.e., SIMD). The present invention also fits into this SIMD category, however, it is further capable of performing scalar concurrent execution for multiple instruction streams (i.e., MIMD) and further for single or multiple context (i.e., SC-MIMD or MC-MIMD).

SIMAC implicitly assigns instructions (termed "machine operations") to processor elements by virtue of the physical ordering of the instructions when they are grouped together into "parallel instructions." The identity of the assigned processor is not a part of the instruction per se. The present invention teaches that the order of the instructions within a group of concurrently executable instructions does not determine the processor assignment. Rather, the processor assignment, under the teachings of the present invention, is explicitly made by adding a logical processor number to become physically part of the instruction which is performed in software prior to execution. In addition, the present invention distinguishes the logical processor number from the actual physical processor number that executes the instruction. This allows for further machine independence of the software in that the physical processor assignment is performed dynamically at execution time.

SIMAC code generation can be divided into three components during which the static allocation and scheduling of its instructions are performed (prior to execution). First, is the determination of the machine operations to be performed. This is the typical code generation phase of a compiler. Second, is the formation of these operations into parallel instructions containing the operations that can be done in parallel. The third component comprises the ordering (scheduling) of the parallel instructions so

> "...to allow the execution of two [parallel instructions] containing different length [machine opera-

tions] to overlap using only simple hardware controls..." McDowell's Conference Paper @ Pg. 475. In other words, McDowell attempts to schedule instructions with differing length execution times so that they can execute concurrently with a minimum amount of hardware support. To the contrary, the present invention does not require this third component. The use of the logical processor number and the instruction firing time that are physically attached to each instruction preclude the need for this phase.

As stated above, SIMAC performs its concurrency detection during compile time only. As a result, the static analysis of the concurrency present within the instruction stream is language dependent. In contrast, the concurrency detection of present invention is language independent since the detection takes place after compilation and prior to execution. The present invention can perform its concurrency detection and scheduling without requiring any modifications to the compiler.

Like SIMAC, the present invention is designed to be independent of the number of processor elements contained within the system. The processor elements in SIMAC, however, are conventional load/store RISC style processors that contain contextual information regarding the state of the previous execution status. These are the "T" and "V" bits associated with each of the SIMAC processor elements. These are used for branching and are the result of executing certain comparison and test instructions. The processor elements of the present invention are context free processors that support a RISC-like instruction set. The term "context free" means that the processor elements contain no state information, e.g., condition codes, registers, program status words, flags, etc. Like SIMAC, the processor elements of the present invention contain no local registers. All operations are performed on data already stored in an array of registers and are accessible by any of the individual processor elements through a register switch.

In SIMAC, only a single set (or level) of 32 registers is available and is shared by all processor elements. Each register may be physically read by any or all processor elements simultaneously. However, only one processor element may write into a specific register at any one time. (Dissertation Thesis at 26) Whereas the registers of this invention also have this characteristic, the architecture of the register set in the present invention is much different than that of the SIMAC machine. The SIMAC machine contains a single level of registers. The present invention contains a number of levels of register sets; each level corresponding to a procedural depth relative to the main program. In addition, each set of levels is replicated to support each context or user that is currently being executed. Thus, under the teachings of the present invention, contexts may be switched or procedures entered without having to flush any registers to memory.

SIMAC does not discuss the need for the production and assignment of condition codes. Without the use of multiple condition code sets, it is not possible to distribute instructions which affect the condition codes across multiple processor elements and guarantee proper condition code data for the execution of subsequent dependent instructions. The present invention teaches a method for the management and assignment of multiple condition code sets. The management and assignment of this resource is analogous to that done for the register resources.

SIMAC also requires a separate control processor that performs the branching operations. This processor contains the program counter and executes the branch operations Each processor element in SIMAC contains three status bits used in the branching operations. The present invention does not use a control processor, nor does it use a program counter and its processor elements contain no such status bits. SIMAC is a centralized control system under the Davis classification whereas the present invention is decentralized.

Hence, the present invention is significantly more general than the SIMAC approach since the present invention performs inter-program parallelism as well as intra-program parallelism.

In addition, while SIMAC provides a register array, McDowell indicates that this, perhaps is not desirable because of cost and proposes that future work be done on an arrangement other than shared registers (Id. at 79) The present invention not only makes use of the shared register concept but also provides sharing of condition codes and the management of those condition codes as well. While McDowell, implicitly, assigns the machine operations to his processor elements, the present invention explicitly extends each instruction with a logical processor number as well as providing a specific firing time for the instruction. There is no disclosure in SIMAC of either a firing time or a logical processor number. Therefore, the master controller, in SIMAC, uses a program counter that provides the control for the processor elements. Under the teaching of the present invention, a set of logical resource drivers (one for each context or user) is provided and instruction selection is based upon the firing times added to each instruction within a basic block (i.e., time driven).

DeSantis sets forth in U.S. Letters Pat. Nos. 4,466,061 and 4,468,736 a concurrent data processor which is:

> adapted to receive strings of object code, form them into higher level tasks and to determine sequences of such tasks which are logically independent so that they may be separately executed ('736 patent at col. 2, lines 50–54).

The logically independent sequences are separately executed by a plurality of processor elements. The first step in the DeSantis approach is to hardware compile the program into strings of object code or machine language in a form which is particularly designed for the computer architecture to provide a dependency free code. The DeSantis invention then forms an independency queue so that all processing for the entire queue is accomplished in one step or cycle. The hardware structure for the DeSantis approach is set forth in FIG. 2 of the patent and contemplates the use of a number of small processor elements (SPEs) each with local memories.

The nature of the interconnection between the local memories of each SPE and the direct storage module **14** is not clear. The patent states:

> In the meantime, respective data items required for execution have been fetched from main memory and stored at the appropriate locations in local memories which locations are accessed by the pointers and the job queue ('736 patent at Col. 6, Lines 19–23).

It does not appear from this description, that each individual SPE has access to the local memories of the other SPE rather, this allocation seems to be made by

the direct storage module (DSM). Hence, under DeSantis, the queue which contains the independent sequences for processing delivers the string to an identified processor element, so that the processor elements can process all strings concurrently and in parallel, the direct storage module must deliver the necessary results into the local memories for each SPE. With respect to each individual SPE, DeSantis states:

> Since the respective processors are provided to execute different functions, they can also be special purpose microprocessors containing only that amount of logic circuitry required to perform the respective functions. The respective circuits are the arithmetic logic unit, shift unit, multiply unit, indexing unit, string processor and decode unit ('736 patent at Col. 7, Lines 7–13).

The primary difference between the approach of the present invention and DeSantis relates to when the detection of the concurrency occurs. TDA performs its concurrency detection during a pre-processing stage at the object level using information normally thrown away by the source code level language processors. DeSantis performs it with part of the hardware collectively called the cache mechanism **10** (see FIG. **1**) dynamically during execution time. Hence, the DeSantis approach constantly uses overhead and resources to hardware de-compile each program. A secondary difference relates to the execution of individual streams of dependent instructions. DeSantis requires that a stream of dependent instructions be executed on the same processor. To the contrary, the present invention permits the execution of a stream of dependent instructions on the same or multiple processor elements.

It is believed that the aforesaid McDowell and DeSantis references are the most pertinent of the references discovered in the patentability investigation and, therefore, more discussion occurs for it than the following references.

J.R. Vanaken in "The Expression Processor" article classifies his processor as a "direct descendant of the tree processor." There is a similarity between the Vanaken architecture and the architecture of the present invention wherein each processor element is capable of accessing, through an alignment network, any one of a number of registers located in a register array. Vanaken utilizes thirty-two processor elements and eight register modules in the register storage. A crossbar switch is disclosed for the alignment network. In this configuration, the identical processor elements of Vanaken do not exchange data directly with the main memory. Rather, data transfers take place between the register storage and the memory. This is not the configuration of the present invention. Vanaken further contemplates utilizing a separate compiler (only for scaler programs) for the detection of low level parallelism and for the assignment of detected parallelism to individual processor elements. How this is done, however, is not disclosed. Vanaken simply discloses the desired goals:

> First, it [the compiler] must detect the potential parallelism in the program. Second, it must map detected parallelism onto the structure of the target machine. *The Expression Processor* at 535.

Vanaken refers only to the parallelism detection techniques presented by Kuck. However, Vanaken's tree structured processor element architecture requires that the computational task be modeled as a computational tree, or as a computational wavefront (wavefront ref: Kuck's work). In the former case, processor assignment is accomplished by assigning a processor to each node in the tree. If there are more nodes in the tree than processors, the task must be broken down into smaller subtasks that will fit on the processor array. In the latter case, the wavefront propagates from the lowest level of processors to the next higher level and so on until the highest level is achieved. Again, if the number of processors required by the wavefront is greater than the number of processors available, the task must be broken into smaller subtasks. This type of flow through ordered processors does not exist in the present invention.

In the Hagiwara articles, the disclosed MIMD QA-1 computer system appears to be a very long instruction word (VLIW) machine having a 160 bit or 80 bit word. The Hagiwara references term this a "horizontal-type microinstruction" and the QA-1 machine takes advantage of the low-level parallel concurrency, at the microcode level. The QA-2 system has 62 working registers constructed of large capacity/high-speed RAM chips interconnected over a network to each of the four processor elements. The earlier QA-1 system utilized a stack of only 15 working registers. Hence, the QA-2 hardware design contemplates the use of individual processor elements having full access to a number of working registers through a switching network. In QA-2, through use of the network, the results of one processor element are delivered to a working register which is then delivered to another processor element as input data. Hence, the QA-1 and QA-2 configurations take advantage of scaler concurrency, multiple ALUs, and the sharing of working registers. The QA-2 approach modifies the QA-1 approach by providing special purpose registers in the register file (i.e., constant, general, indirect, and special); such a division, however, is not apparent in the QA-1 architecture. From a compiling point of view, the QA-1 was primarily designed to be encoded by a programmer at the microcode level to take programming advantage of the inherent power of the system's architecture. The QA-2 machine which is directed towards the personal computer or local host computer (e.g., for laboratory use) contemplates that programs written in high-level languages will be compiled into the QA-2 microprogrammed interpretations.

The four Fisher references disclose a SIMD system based upon a "very long instruction word" (VLIW). The goal presented in these articles is to determine the low level parallelism or "fine-grained parallelism" of a program in a separate compiler so that all individual movements of the data within the VLIW machine are completely specified at compile time. At the outset, it is important to recognize that the hardware architecture for implementing Fisher's VLIW machine is not discussed in these references other than in a general sense. Fisher contemplates:

> Each of the eight processors contain several functional units, all capable of initiating an operation in each cycle. Our best guess is two integer ALUs, one pipelined floating ALU, one memory port, several register banks, and a limited crossbar for all of these to talk to each other. *Computer* article at pg. 50.

Rather, the Fisher articles concentrate on the software compiling technique called "trace scheduling" and nicknamed BULLDOG. Fisher further requires an instruction word of "fixed length." The length of Fisher's instruction word, therefore, amount of hardware required to process the instruction word. In that respect, the present invention is much more dynamic and fluid

since it can have any number of processors wherein each processor processes the parallel instruction during the instruction firing time.

Lorie et al. (U.S. Letters Pat. No. 4,435,758) sets forth a technique for ordering instructions for parallel execution during compile time by adding code to the instruction stream. The present invention does not add code to the instruction stream.

The Reigel patent (U.S. Letters Pat. No. 3,611,306) relates to the ordering of the detected parallel instructions for a particular computer architectural approach.

The patent issued to Freiman, et al. (U.S. Letters Pat. No. 3,343,135) teaches a dynamic compiling system, hardware based, for use in a multiprocessor environment for analyzing particular types of mathematical expressions for parallel execution opportunities and then to automatically assign individual operations to processors within the system. Freiman first identifies the concurrencies, in the hardware, and determines the time periods within which the operations may be performed. These determined times are "the earliest times in which a given operation may be performed." The Freiman invention also analyzes the particular mathematical expression and determines, in the hardware, "the latest times in which an operation can be performed." This information generated from the analysis of the mathematical algorithm is stored in a word register which is then used in the multiprocessor environment. Each processor has associated with it a processor control which for all processors is identical and which is driven by six control fields. One function of the control block is to determine whether or not a particular processor is ready to receive a job, to indicate when it has received a job, and to analyze each job to see if the processor can proceed immediately with the job or whether it must wait until one of the operants of the job is already assigned. Once a processor is assigned to a task the necessary data contained in the results register may not be available at that particular time and hence the processor must wait until a test of the results register indicates that the result is available. Once the result is available, the processor will perform its operation and place its result in the appropriate result register for access by another processor. All processors in Freiman have access to the results register by means of a crossbar switch.

The patent issued to Dennis et al. (U.S. Letters Pat. No. 4,153,932) teaches a highly parallel multiprocessor for the concurrent processing of programs represented in data flow form. Specifically, a mathematical computation as shown at column 5, lines 17–23 and as represented in FIG. 2 in data flow. language is used as an example. Hence, the data flow form is generated and stored in the memory of the processor in designated instruction cells wherein each instruction cell corresponds to an operator in the data flow program. In operation, when an instruction cell is complete (i.e., containing the instruction and all necessary operands), a signal is generated to the Arbitration Network which directs the flow of the information in the instruction cell to the identified processor. The processor operates on the information and delivers it back through a distribution network into the main memory.

The patent issued to Culler (U.S. Letters Pat. No. 3,771,141) discloses a high speed processor capable of parallel operations on data. Multiple or parallel processors, however, are not used. Rather, the parallelism is achieved structurally by implementing each of the processor's data registers with four separate multi-bit data input ports. Hence, four operations can be performed at once. This results in a "tightly coupled" arrangement since the data from any given processor register is ready to be received by any other register at the next clock cycle. The control of which register is to receive which information occurs at the object code level and, as shown in Table 1 of Culler, four additional fields are provided in each object code instruction for controlling the flow of data among the processor's registers. Culler states:

> The data pad memory is tightly coupled to the arithmetic unit and serves as an effective buffer between the high speed arithmetic unit and a large capacity randam access core memory. As an indication of the effective speed, a complete multiplication of two signed eight bit words can be accomplished in three cycles or 375 nanoseconds.

The Gilliland et al. patent (U.S. Letters Pat. No. 4,229,790) sets forth a processor for processing different and independent jobs concurrently through the use of pipelining with precedent constraints. The Gilliland invention is capable of processing programming tasks concurrently as well as processing the parallel parts of each programming task concurrently. It appears that the invention operates on concurrencies in the program at the source code level rather than at the object code level. The Gilliland processor is capable of processing up to 128 parallel processor paths.

The patent issued to Blum et al. (U.S. Letters Pat. No. 4,109,311) relates to a data processing system based upon time slice multiprogramming. The Blum invention contemplates a conventional multiprocessor arrangement wherein a first processor controls a keyboard and display interface, a second processor controls printer, disk storage, and tape storage interface, and a third processor executes the problem oriented programs located in the main storage.

Gruner (U.S. Letters Pat. No. 4,104,720) sets forth a CPU control multiprocessor system wherein each parallel processor is directly interfaced through an interface control circuit to the CPU. Each interface circuit contains memory for storing portions of the micro instructions contained within the CPU. Gruner calls this an "extension" of the micro instructions from the CPU into the interface circuit for each parallel processor. The Gruner system while controlling each interface circuit to allocate and to assign concurrent tasks utilizes a conventional bus structure for the transfer of information between the processors.

The patent issued to Caril (U.S. Letters Pat. No. 4,241,398) relates to a supervisory control system wherein a central processing unit controls and supervises the operation of a number of remote processing units. The Caril invention relates to a low overhead line protocol for controlling the asynchronous exchange of communication information between the central processing unit and each remote processing unit.

Bernhard et al. (U.S. Letters Pat. No. 4,228,495) relates to a multiprocessor numerical control system. The Bernhard invention relates to a main processor coupled over a bus structure to a plurality of programmable interface processors.

Koehler et al. patent (U.S. Letters Pat. No. 4,270,167) discloses a multiprocessor arrangement wherein a central processor has primary control and access to a local bus and wherein the remaining plurality of processors also share access to the local bus. Arbitration among the

processors is provided to the local bus as well as the generation of query status and processor status signals.

The patent issued to Kober (U.S. Letters Pat. No. 4,181,936) relates to a distributed computing system for increasing the efficiency of the data bus.

The Requa article discusses a piecewise data flow architecture which seeks to combine the strengths of other supercomputers. The goal of the Requa architecture is to take the source code program and recompile it into "basic blocks." These basic blocks are designed to be no longer than 255 instructions and the natural concurrencies existing in each basic block are identified and processed separately by a number of "scaler processors." The natural concurrencies are encoded as part of the instructions to the scaler processors. The concurrent instructions waiting for execution reside in a stack of registers to which all of the scaler processors has access to deliver data into and to receive data from. As in the Dennis approach, when a particular instruction residing in the register is completed (i.e., when a prior data dependency has been satisfied and stored) the scaler instruction apparently raises a flag which will be detected for delivery to a scaler processor.

In the Bernhard article, the author surveys the current state of art concerning supercomputer design. The article discusses the current supercomputer research projects and provides critical comments concerning each type of design. The disclosure in this article appears to be cumulative to the analysis set forth above and none of the approaches set forth suggest the TDA processor register communication technique.

SUMMARY OF INVENTION

The present invention provides a method and a system that is non-Von Neuman and one which is adaptable for use in single or multiple context SISD, SIMD, and MIMD configurations. The method and system is further operative upon a myriad of conventional programs without user intervention.

The present invention statically determines at a very fine level of granularity, the natural concurrencies in the basic blocks (BBs) of programs at essentially the object code level and adds a logical processor number (LPN) and an instruction firing time (IFT) to each instruction in each basic block in order to provide a time driven decentralized control. The detection and low level scheduling of the natural concurrencies and the adding of the intelligence occurs only once for a given program after conventional compiling and prior to execution. At this time all instruction resources are assigned.

The present invention further executes the basic blocks containing the added intelligence on a system containing a plurality of identical processor elements each of which does not retain execution state information from prior operations. Hence, all processor elements are context free. Instructions are selected for execution based on instruction firing time. Each processor element is capable of executing instructions on a perinstruction basis such that dependent instructions can execute on the same or different processor elements. A given processor element in the present invention is capable of executing an instruction from one context followed by an instruction from another context. All context information necessary for processing a given instruction is contained elsewhere in the system.

The system and method of the present invention are described in the following drawing and specification.

DESCRIPTION OF THE DRAWING

FIG. **1** is the generalized flow of the TOLL software of the present invention;

FIG. **2** is a graphic representation of a sequential series of basic blocks found within the conventional compiler output;

FIG. **3** is a graphical presentation of the extended intelligence added to each basic block under the teachings of the present invention;

FIG. **4** is a graphical representation showing the details of the extended intelligence added to each instruction within a given basic block;

FIG. **5** is the breakdown of the basic blocks carrying the extended information into discrete execution sets;

FIG. **6** is a block diagram presentation of the architectural structure of the present invention;

FIG. **7***a*–**7***c* represents an illustration of the network interconnections during three successive instruction firing times;

FIGS. **8**–**11** are the flow diagrams setting forth the program features of the software of the present invention;

FIG. **12** is the flow diagram for determining the execution sets in the TOLL Software;

FIG. **13** sets forth the register file organization of the present invention;

FIG. **14** illustrates the transfers between registers in levels during a subroutine call;

FIG. **15** sets forth the structure of the logical resource drivers (LRDs) of the present invention;

FIG. **16** sets forth the structure of the instruction caches control and of the caches of the present invention;

FIG. **17** sets forth the structure of the PIQ buffer unit and the PIQ bus interface unit of the present invention;

FIG. **18** sets forth interconnection of the processor elements through the PE-LRD network to the PIQ processor alignment circuit of the present invention;

FIG. **19** sets forth the structure of the branch execution unit of the present invention;

FIG. **20** illustrates the organization of the contexts of the present invention;

FIG. **21** sets forth the structure of one embodiment of the processor element of the present invention; and

FIGS. **22**(*a*) through **22**(*d*) sets forth the data structures in the processor element of FIG. **21**.

GENERAL DESCRIPTION

1. Introduction

In the following two sections, a general description of the software and hardware of the present invention takes place. The system of the present invention is designed based upon a unique relationship between the hardware and software components. While many prior art approaches have primarily provided for multiprocessor parallel processing based upon a new architecture design or upon unique software algorithms, the present invention is based upon a unique hardware/software relationship. The software of the present invention provides the intelligent information for the routing and synchronization of the instruction streams through the hardware. In the performance of these tasks, the software spatially and temporally manages all user accessible resources, e.g., registers, condition codes, memory and stack pointers. This routing and synchronization is done without any user intervention, and does not require changes to the source code. Additionally, the